Volatility timing

We're going to look at how to manage portfolio volatility. Managing volatility is a fundamental risk-management task. You probably have some notion of the amount of volatility that you want in your portfolio, or that you feel comfortable bearing. Maybe you're ok with an annualized volatility of 15%, but 30% is too much and makes it hard for you to sleep at night. If that's the case, then when markets get volatile you might want to take action to reduce your portfolio's volatility. We'll discuss some strategies for predicting and managing volatility below.

We will focus on allocating between a risky asset and a risk-free asset as a way to manage volatility (i.e., two-fund seperation). We're taking the risky asset, such as the market portfolio of equities, as given. The essential idea is to put more weight on the risky asset when expected volatility is low and less weight on the risky asset when expected volatility is high. This is related to a portfolio construction strategy known as risk-parity. The managed volatility strategy that we consider below is based off work by Barroso and Santa-Clara (2015), Daniel and Moskowitz (2016), and Moreira and Muir (2017). The Moreira and Muir (2017) paper is probably the best place to start. Though the observation that a) predictable volatility and b) unpredictable returns implies predictable Sharpe ratios predates the aoove work.

Acquiring Fama-French data

As a start, let's acquire a long daily time series on aggregate US market returns. We'll use the 3 factors dataset.

We're going to use System.IO.Compression

```
#r "nuget: FSharp.Data"
open System
open System.Net
open System.IO
open System.IO.Compression
open FSharp.Data

/// URL for the Fama and French 5 factor zip archive
let cacheDirectory = Path.Combine(__SOURCE_DIRECTORY__,"../data-cache")
let ff3Url = "http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/F-F_Research_Data_Factors_daily_CSV.zip"
/// Local file path for the Fama and French 5 factor zip archive
let ff3ZipFilePath = Path.Combine(cacheDirectory,"ff3.zip")

let web = new WebClient()

web.DownloadFile(ff3Url, ff3ZipFilePath)
web.Dispose()
// inspect entries
let ff3ZipArchive = ZipFile.OpenRead(ff3ZipFilePath)
ff3ZipArchive.Entries
/// Local csv for Fama and French 5 factor data
let ff3Csv =
    ff3ZipArchive.Entries
    |> Seq.head
    |> fun entry -> Path.Combine(cacheDirectory,entry.Name)


// Extract entry
if FileInfo(ff3Csv).LastWriteTime < (DateTime.Now.AddDays(-20.0)) then
    File.Delete(ff3Csv)
ff3ZipArchive.ExtractToDirectory(cacheDirectory)
ff3ZipArchive.Dispose() // we're done with the archive

let fileLines = File.ReadAllLines(ff3Csv)

fileLines
|> Seq.take 5
|> Seq.toList


fileLines
|> Seq.skipWhile(fun line -> not (line.Contains("Mkt-RF")))
|> Seq.takeWhile(fun line -> line <> "")

// CsvType
type FF3Csv = CsvProvider<"Date (string),Mkt-RF,SMB,HML,RF
19260701,    0.10,   -0.24,   -0.28,   0.009
19260702,    0.45,   -0.32,   -0.08,   0.009
19260706,    0.17,    0.27,   -0.35,   0.009
19260707,    0.09,   -0.59,    0.03,   0.009
19260708,    0.21,   -0.36,    0.15,   0.009">

type FF3Obs =
    { Date : DateTime
      MktRf : float
      Smb : float
      Hml : float
      Rf : float }
```

Some info on date parsing here. But if we were ever going to save this date data to disk or work with times, we should prefer DateTimeOffset or NodaTime.

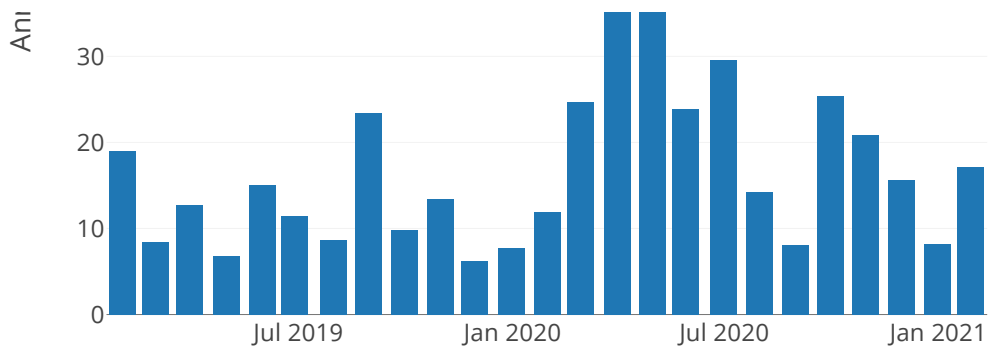
```
let dateParse x =
    DateTime.ParseExact(x,
        "yyyyMMdd",
        Globalization.CultureInfo.InvariantCulture)
dateParse("20200104")

let ff3 =
    fileLines
    |> Array.skipWhile(fun line -> not (line.Contains("Mkt-RF")))
    |> Array.skip 1
    |> Array.takeWhile(fun line -> line <> "")
    |> Array.map(fun line ->
        let parsedLine = FF3Csv.ParseRows(line).[0]
        { Date = dateParse parsedLine.Date
          MktRf = float parsedLine.``Mkt-RF`` / 100.0
          Smb = float parsedLine.SMB / 100.0
          Hml = float parsedLine.HML / 100.0
          Rf = float parsedLine.RF / 100.0 })


fsi.AddPrinter<DateTime>(fun dt -> dt.ToString("s"))

ff3 |> Seq.take 5
```

Observing time-varying volatility

One thing that can help us manage volatility is the fact that volatility tends to be somewhat persistent. By this we mean that if our risky asset is volatile today, then it is likely to be volatile tomorrow. We can observe this by plotting monthly volatility as we do below. It also means that we can use past volatility to form estimates of future volatility.

```
#r "nuget: FSharp.Stats, 0.4.0"
#r "nuget: Plotly.NET, 2.0.0-beta5"
open FSharp.Stats
open Plotly.NET


let annualizeDaily x = x * sqrt(252.0) * 100.

let monthlyVol =
    ff3
    |> Seq.sortBy(fun x -> x.Date)
    |> Seq.groupBy(fun x -> x.Date.Year, x.Date.Month)
    |> Seq.map(fun (_ym, xs) ->
        let dt = xs |> Seq.last |> fun x -> x.Date
        let annualizedVolPct = xs |> stDevBy(fun x -> x.MktRf) |> annualizeDaily
        dt, annualizedVolPct)
    |> Seq.toArray

let volChart vols =
    let getYear f = vols |> Seq.map(fun (dt:DateTime,_vol) -> dt.Year ) |> f
    let minYear = getYear Seq.min
    let maxYear = getYear Seq.max
    vols
    |> Chart.Column
    |> Chart.withTraceName $"Time-varying Volatility ({minYear}-{maxYear})"
    |> Chart.withY_AxisStyle(title = "Annualized Volatility (%)")

let allVolsChart = volChart monthlyVol
let since2019VolChart =
    monthlyVol
    |> Seq.filter(fun (dt,_) -> dt >= DateTime(2019,1,1))
    |> volChart
```

```
allVolsChart |> Chart.Show
since2019VolChart |> Chart.Show
```



Review of calculating portfolio weights

We are going to look at various portfolios, so it is good to review portfolio weights.

- Portfolio weight is $(\text{position value})/(\text{portfolio value})$.
- A long portfolio has portfolio weights that sum to 1.0 (or 100%).
- A zero-cost portfolio has portfolio weights that sum to 0.0.

```
type Position = { Id: string; Position: int; Price : decimal }

let portfolio =
    [| { Id = "AAPL"; Position = 100; Price = 22.20m }
       { Id = "AMZN"; Position = 20; Price = 40.75m }
       { Id = "TSLA"; Position = 50; Price = 30.6m } |]

// We can do it on the fly
portfolio
|> Array.map(fun pos -> // getting position value
       pos.Id,
       (float pos.Position)*(float pos.Price))
|> fun ps -> // calculating weights
    let portfolioValue =
        ps
        |> Array.sumBy(fun (id, value) -> value)
    ps
    |> Array.map(fun (id, value) -> id, value / portfolioValue)


// We can also define a bit more structure to get the same thing.
// Proper functions with defined types are good for reusable code.
type PositionValue = { Id : string; Value : float }
type PositionWeight = { Id : string; Weight : float }

let calcValue (x:Position) =
    { Id = x.Id
      Value = (float x.Position) * (float x.Price)}

let calcWeights xs =
    let portfolioValue = xs |> Array.sumBy(fun x -> x.Value)
    xs
    |> Array.map(fun pos ->
        { Id = pos.Id
          Weight = pos.Value / portfolioValue })

portfolio
|> Array.map calcValue
|> calcWeights

let portfolioWithShorts =
    [| { Id = "AAPL"; Position = 100; Price = 22.20m }
       { Id = "AMZN"; Position = -20; Price = 40.75m }
       { Id = "TSLA"; Position = 50; Price = 30.6m } |]

portfolioWithShorts
|> Array.map calcValue
|> calcWeights
```

Effect of leverage on volatility

Recall the formula for variance of a portfolio consisting of stocks x and y:

$$\sigma^2 = w_x^2\sigma_x^2 + w_y^2\sigma_y^2 + 2w_xw_ycov(r_x, r_y),$$

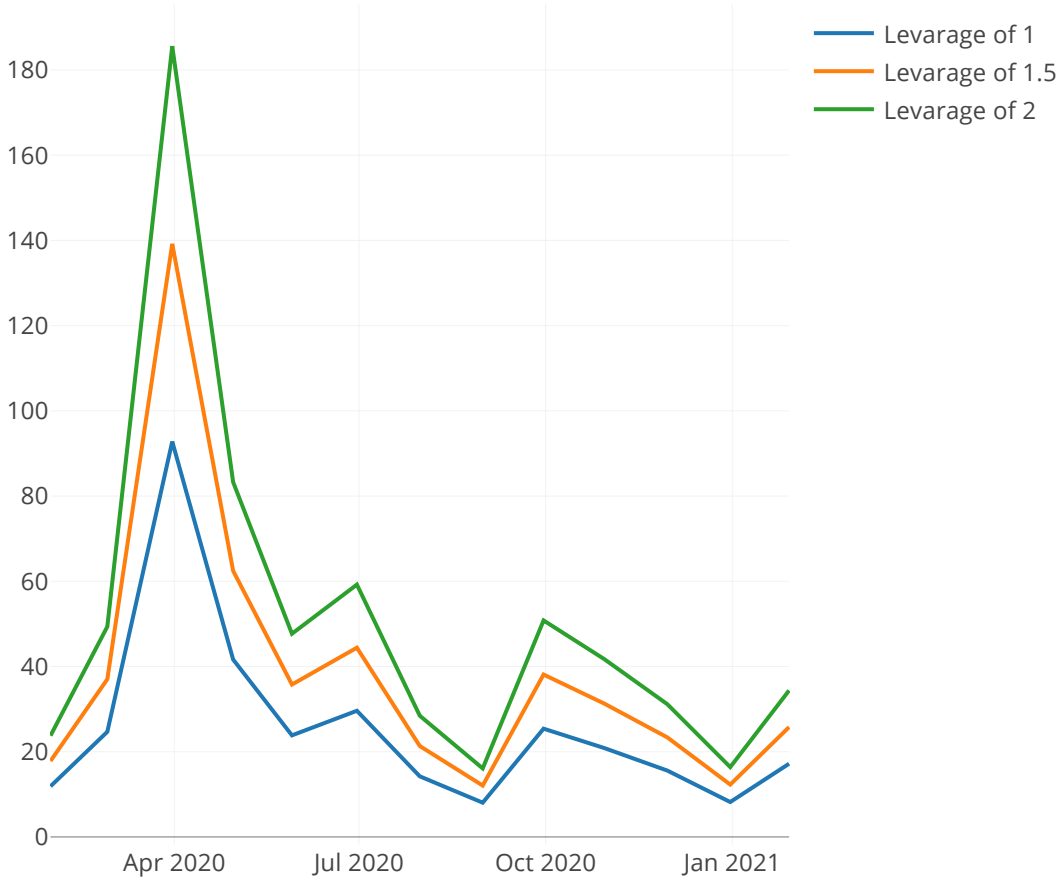
where w_x and w_y are the weights in stocks x and y, r_x and r_y are the stock returns, σ^2 is variance, and $cov(.,.)$ is covariance.

If one asset is the risk free asset (borrowing a risk-free bond), then this asset has no variance and the covariance term is 0.0. Thus we are left with the result that if we leverage asset x by borrowing or lending the risk-free asset, then our leveraged portfolio's standard deviation (σ) is

$$\sigma^2 = w_x^2\sigma_x^2 \rightarrow \sigma = w_x\sigma_x$$

```fsharp
let leveragedVol (weight, vol) = weight * vol

/// We're doing leverage in terms of weigh on the risky asset.
/// 1 = 100%, 1.25 = 125%, etc.
let exampleLeverages =
    [ 1.0; 1.5; 2.0 ]

let rollingVolSubset =
    monthlyVol
    |> Seq.filter(fun (dt, vol) -> dt > DateTime(2020,1,1))

let exampleLeveragedVols =
    exampleLeverages
    |> Seq.map(fun leverage ->
        leverage,
        rollingVolSubset
        |> Seq.map(fun (dt, vol) -> dt, leveragedVol(leverage, vol)))

let exampleLeveragesChart =
    exampleLeveragedVols
    |> Seq.map(fun (leverage, leveragedVols) ->
        leveragedVols
        |> Chart.Line
        |> Chart.withTraceName $"Levarage of {leverage}")
    |> Chart.Combine
```

```fsharp
exampleLeveragesChart |> Chart.Show
```



Effect of leverage on returns

The effect of leverage on returns can be seen from the portfolio return equation,

$r_p = \Sigma_{i=1}^{N} w_i r_i,$

where r is return, i indexes stocks, and w is portfolio weights.

So if we borrow 50% of our starting equity by getting a risk-free loan, then we have

$r_{\text{levered}} = 150\% \times r_{\text{unlevevered}} - 50\% \times r_f$

If we put in terms of excess returns,

$r_{\text{levered}} - r_f = 150\% \times (r_{\text{unlevevered}} - r_f) - 50\% \times (r_f - r_f) = 150\% \times (r_{\text{unlevevered}} - r_f)$

So, if we work in excess returns we can just multiply unlevered excess returns by the weight.

Does this check out? Imagine that you have $1 and you borrow $1 for a net stake of $2. Then you invest at an exess return of 15%. What are you left with?

```
let invest = 1.0m
let borrow = 1.0m
let ret = 0.15m
let result = (invest + borrow)*(1.0m+ret)-borrow
result = 1.0m + ret * (invest + borrow)/invest

let leveredReturn leverage (x : FF3Obs ) = leverage * x.MktRf
```

We can illustrate this with cumulative return plots. Let's first show a simple example for how we can calculate cumulative returns.

Imagine that you invest $1 at a 10% return. What do you have after n years?

```
[| 1.0 .. 5.0 |]
|> Array.map(fun years -> 1.0*(1.1**years))
```

But what if we have the data like this?

```
[| for i = 1 to 5 do 0.1 |]
```

For this, we could use a scan or a mapFold. The website www.fsharpforfunandprofit.com has a good discussion of the various module functions. I think a mapFold is a bit easier in this circumstance, so let's go with that.

If we look at the definition of `mapFold`, we can see that we call it with `Array.mapFold mapping state array`, where

```
mapping : 'State -> 'T -> 'Result * 'State
The function to transform elements from the input array and accumulate the final value.

state : 'State
The initial state.

array : 'T[]
The input array.

Returns: 'Result[] * 'State
The array of transformed elements, and the final accumulated value.
```

```
let exampleMapper inCumRet ret =
    let outCumRet = inCumRet * (1.0 + ret)
    outCumRet, outCumRet

let exampleMapper2 inCumRet ret =
    let outCumRet = inCumRet * (1.0 + ret)
    outCumRet - 1.0, outCumRet

let exampleInitialState = 1.0 // your $1 at the start
let exampleArrayToAccumulate = [| for i = 1 to 5 do 0.1 |]
Array.mapFold exampleMapper exampleInitialState exampleArrayToAccumulate
```

```
val exampleMapper : inCumRet:float -> ret:float -> float * float
val exampleMapper2 : inCumRet:float -> ret:float -> float * float
val exampleInitialState : float = 1.0
val exampleArrayToAccumulate : float [] = [|0.1; 0.1; 0.1; 0.1; 0.1|]
val it : float [] * float = ([|1.1; 1.21; 1.331; 1.4641; 1.61051|], 1.61051)
```

```
// or
exampleArrayToAccumulate |> Array.mapFold exampleMapper exampleInitialState
```

```
val it : float [] * float = ([|1.1; 1.21; 1.331; 1.4641; 1.61051|], 1.61051)
```

```
// Relative to a 0 starting point.
exampleArrayToAccumulate |> Array.mapFold exampleMapper2 exampleInitialState
```

```
val it : float [] * float = ([|0.1; 0.21; 0.331; 0.4641; 0.61051|], 1.61051)
```

That's a cumulative return! We can get rid of the cumulative state in the second part of the tuple with a `fst` call since it is a tuple of pairs.

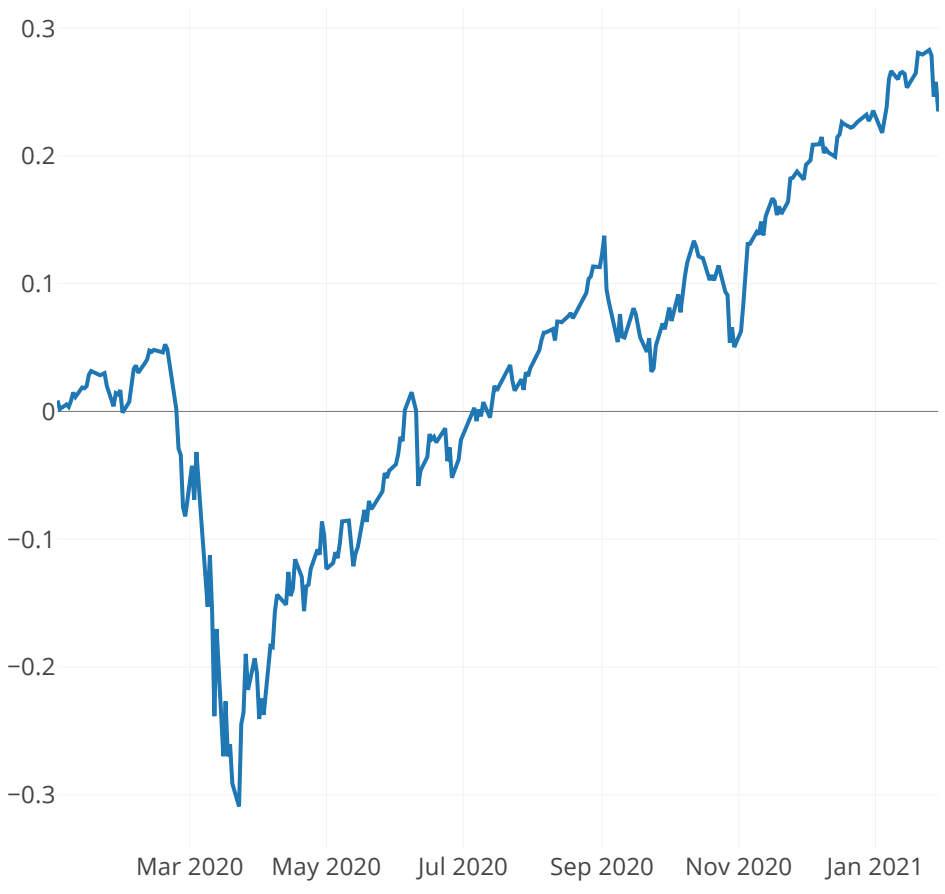
```
exampleArrayToAccumulate
|> Array.mapFold exampleMapper2 exampleInitialState
|> fst
```

```
val it : float [] = [|0.1; 0.21; 0.331; 0.4641; 0.61051|]
```

Now we know how to calculate cumulative returns on our real data.

```
let since2020 =
    ff3 |> Seq.filter(fun x -> x.Date >= DateTime(2020,1,1))

let cumulativeReturnEx =
    let mapping inCumRet x =
        let outCumRet = inCumRet * (1.0 + x.MktRf)
        { x with MktRf = outCumRet - 1.0}, outCumRet

    since2020
    |> Seq.mapFold mapping 1.0
    |> fst

let cumulativeReturnExPlot =
    cumulativeReturnEx
    |> Seq.map(fun x -> x.Date.Date, x.MktRf)
    |> Chart.Line
```

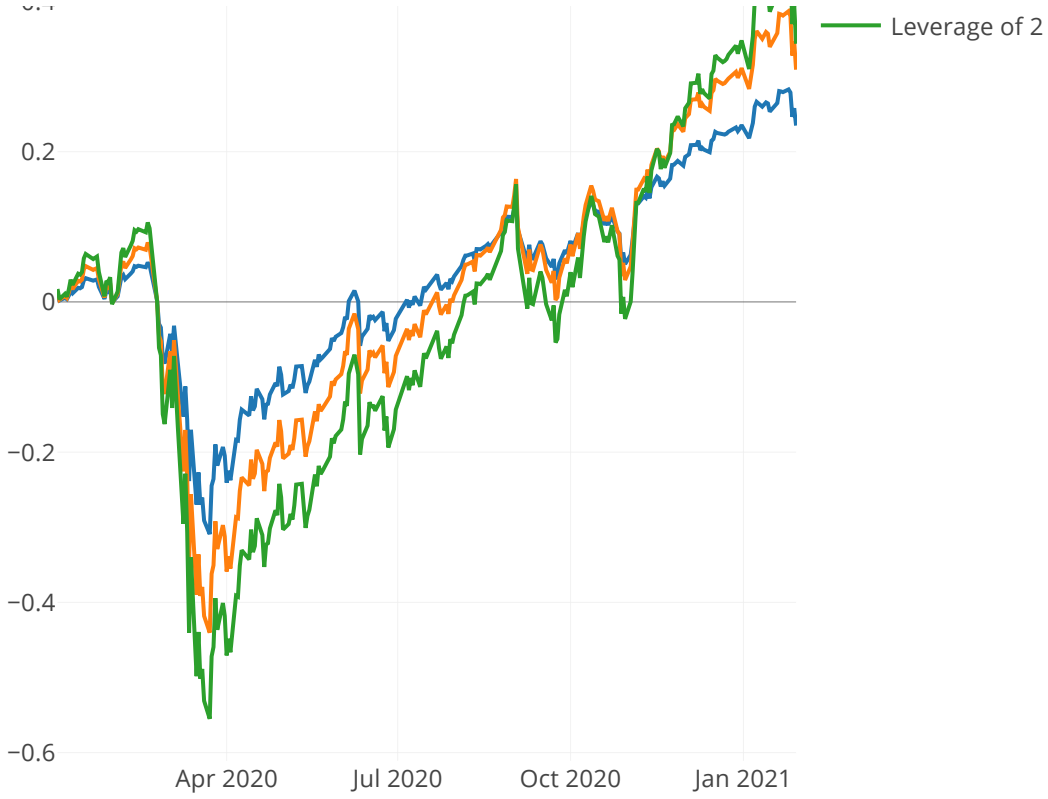
```
cumulativeReturnExPlot |> Chart.Show
```



Let's try leverage with daily rebalancing (each day we take leverage of X).

```
let getLeveragedReturn leverage =
    let mapping inCumRet x =
        let lr = leveredReturn leverage x
        let outCumRet = inCumRet * (1.0 + lr)
        { x with MktRf = outCumRet - 1.0}, outCumRet

    since2020
    |> Seq.mapFold mapping 1.0
    |> fst
    |> Seq.map(fun x -> x.Date.Date, x.MktRf)

let exampleLeveragedReturnChart =
    exampleLeverages
    |> Seq.map(fun lev ->
        getLeveragedReturn lev
        |> Chart.Line
        |> Chart.withTraceName $"Leverage of {lev}")
    |> Chart.Combine
```

```
exampleLeveragedReturnChart |> Chart.Show
```



Predicting volatility

To manage or target volatility, we need to be able to predict volatility. A simple model would be to use past volatility to predict future volatility. How well does this work?

Let's try sorting days into 5 groups based on trailing 20-day volatility, then we'll see if this sorts actual realized volatility.

```
let dayWithTrailing =
    ff3
    |> Seq.sortBy(fun x -> x.Date)
    |> Seq.windowed 23
    |> Seq.map(fun xs ->
        let train = xs |> Array.take (xs.Length-1)
        let test = xs |> Array.last
        train, test)

dayWithTrailing
|> Seq.sortBy(fun (train, _test) -> train |> stDevBy(fun x -> x.MktRf))
|> Seq.splitInto 5
|> Seq.iter(fun xs ->

    let predicted =
        xs
        |> Array.collect fst
        |> stDevBy (fun x -> x.MktRf)
        |> annualizeDaily
    let actual =
        xs
        |> Array.map(fun (_train, test) -> test.MktRf)
        |> stDev
        |> annualizeDaily
    printfn $"N: {xs.Length}, Predicted: %.1f{predicted}, Actual: %.1f{actual}")
```

```
N: 4979, Predicted: 6.5, Actual: 8.5
N: 4979, Predicted: 9.0, Actual: 10.1
N: 4979, Predicted: 11.2, Actual: 12.1
N: 4978, Predicted: 15.1, Actual: 15.5
N: 4978, Predicted: 31.4, Actual: 30.1
```

Even with this very simple volatility model, we seem to do a reasonable job predicting volatility. We get a decent spread.

Targetting volatility

How can we target volatility? Recall our formula for a portfolio's standard deviation if we're allocating between the risk-free asset and a risky asset:

$\sigma = w_x \sigma_x$

If we choose

$$w_x = \frac{\text{target volatility}}{\text{predicted volatility}} = \frac{\text{target}}{\hat{\sigma}_x}$$

then we get

$$\sigma = \frac{\text{target} \times \sigma_x}{\hat{\sigma}_x}$$

We're not going to predict volatility perfectly, but *if* we did then we'd have a portfolio that had a constant volatility equal to our target.

Let's see what happens if we try to target 15% annualized volatility.

```
type VolPosition =
    { Date : DateTime
      Return : float
      Weight : float }

let targetted =
    dayWithTrailing
    |> Seq.map(fun (train,test) ->
        let predicted = train |> stDevBy(fun x -> x.MktRf) |> annualizeDaily
        let w = (15.0/predicted)
        { Date = test.Date
          Return = test.MktRf * w
          Weight = w })

let targettedSince2019 =
    targetted
    |> Seq.filter(fun x -> x.Date >= DateTime(2019,1,1) )
    |> Seq.groupBy(fun x -> x.Date.Year, x.Date.Month)
    |> Seq.map(fun (_, xs) ->
        xs |> Seq.map(fun x -> x.Date) |> Seq.max,
        xs |> stDevBy(fun x -> x.Return) |> annualizeDaily)
    |> volChart
    |> Chart.withTraceName "Managed"

let rawSince2019 =
    monthlyVol
    |> Seq.filter(fun (dt,_) -> dt >= DateTime(2019,1,1))
    |> volChart
    |> Chart.withTraceName "Unmangaged"

let weightsSince2019 =
    targetted
    |> Seq.filter(fun x -> x.Date >= DateTime(2019,1,1) )
    |> Seq.groupBy(fun x -> x.Date.Year, x.Date.Month)
    |> Seq.map(fun (_, xs) ->
        xs |> Seq.map(fun x -> x.Date) |> Seq.max,
        xs |> Seq.averageBy(fun x -> x.Weight))
    |> Chart.Line
    |> Chart.withTraceName "weight on the Market"

let volComparison =
    [ targettedSince2019; rawSince2019]
    |> Chart.Combine

let volComparisonWithWeights =
    Chart.Grid([[volComparison]
                [weightsSince2019 ]],
                sharedAxes = true)
```

```
volComparisonWithWeights |> Chart.Show
```

We can see that in this example,

- Volatility still moves around with our managed portfolio. We haven't targetted a 15\% volatility perfectly.
- We do avoid some of the extreme volatilities from 2020, particularly in March and April.
- The weight we put on the market varies quite (implies lots of trading) and goes as high as 2 (lots of leverage).

Evaluating performance.

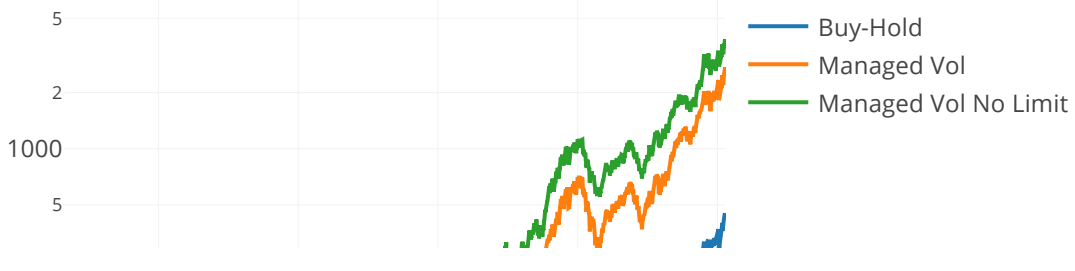
Let's now compare buy and hold to our managed volatility strategy.

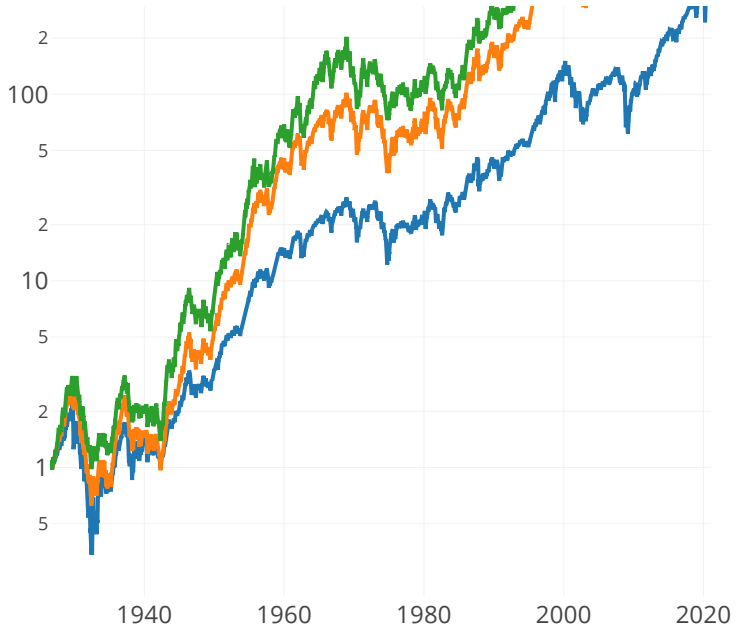
For the managed portfolio, we'll also impose the constraint that the investor cannot borrow more than 30% of their equity (i.e, max weight = 1.3).

We start by defining functions to calculate the weights. In the managed weights, we set the numerator so that we're targetting the full-sample standard deviation of the market.

```
let leverageLimit = 1.3

let sampleStdDev =
    dayWithTrailing
    |> Seq.stDevBy(fun (train, test) -> test.MktRf)
    |> annualizeDaily

let buyHoldWeight predictedStdDev = 1.0

let inverseStdDevWeight predictedStdDev =
    min (sampleStdDev / predictedStdDev) leverageLimit

let inverseStdDevNoLeverageLimit predictedStdDev =
    sampleStdDev / predictedStdDev


let getManaged weightFun =
    dayWithTrailing
    |> Seq.map(fun (train,test) ->
        let predicted = train |> stDevBy(fun x -> x.MktRf) |> annualizeDaily
        let w = weightFun predicted
        { Date = test.Date
          Return = test.MktRf * w
          Weight = w })
    |> fun xs -> // Rescale to have same realized SD for
                 // more interpretable graphs.
                 // Does not affect sharpe ratio
        let sd = xs |> stDevBy(fun x -> x.Return) |> annualizeDaily
        xs |> Seq.map(fun x -> { x with Return = x.Return * (sampleStdDev/sd)})

let accVolPortReturn (port: seq<VolPosition>) =
    let mapper acc (x : VolPosition) =
        let outAcc = acc * (1.0+x.Return)
        { x with Return = outAcc }, outAcc
    port
    |> Seq.mapFold mapper 1.0
    |> fst

let portChart name port  =
    port
    |> Seq.map(fun x -> x.Date, x.Return)
    |> Chart.Line
    |> Chart.withTraceName name
    |> Chart.withY_Axis (Axis.LinearAxis.init(AxisType = StyleParam.AxisType.Log))


let buyHoldMktPort = getManaged buyHoldWeight
let managedMktPort = getManaged inverseStdDevWeight
let managedMktPortNoLimit = getManaged inverseStdDevNoLeverageLimit

let bhVsManagedChart =
    Chart.Combine(
        [ buyHoldMktPort |> accVolPortReturn |> (portChart "Buy-Hold")
          managedMktPort |> accVolPortReturn |> (portChart "Managed Vol")
          managedMktPortNoLimit |> accVolPortReturn |> (portChart "Managed Vol No Limit")
          ])
```

```
bhVsManagedChart |> Chart.Show
```






```
[ buyHoldMktPort, "Buy-Hold Mkt"
  managedMktPort, "Managed Vol Mkt"
  managedMktPortNoLimit, "Manage Vol No Limit"]
|> Seq.iter(fun (x, name) ->
    let mu =
        x
        |> Seq.averageBy(fun x -> x.Return)
        |> fun x -> round 2 (100.0*252.0*x)
    let sd = x |> Seq.stDevBy(fun x -> x.Return) |> annualizeDaily
    printfn $"Name: %25s{name} Mean: %.2f{mu} SD: %.2f{sd} Sharpe: %.3f{round 3 (mu/sd)}")
```

```
Name:              Buy-Hold Mkt Mean: 7.62 SD: 17.12 Sharpe: 0.445
Name:           Managed Vol Mkt Mean: 9.43 SD: 17.12 Sharpe: 0.551
Name:       Manage Vol No Limit Mean: 9.79 SD: 17.12 Sharpe: 0.572
```

Things to consider

- What's a reasonable level of volatility to target?
- What's a reasonable level of leverage, and what should we think about when leveraging a portfolio that has low recent volatility?
- What happens if expected returns go up when volatility is high?
- How do we decide on what risky portfolio to invest in?